On the Program Schemata with Meta-Flow (プログラムの基礎理論)

AUTHOR(S):

SHA, A.; OKINAKA, I.; KAMBAYASHI, Y.

数理解析研究所講究録
119巻 1971年 1-27

ON THE PROGRAM SCHEMATA WITH META-FLOW

A.SHA (KYOTO UNIV.)
I.OKINAKA (KYOTO UNIV.)
Y.KAMBAYASHI (UNIV. OF ILL.)

§ 1 Introduction

According to the different programming techniques, the structure of programs will become different even if they have the same algorithm. Principally the programming techniques are used to control dynamically the flow of computation during execution. It is natural to think that the segments, the flow of which is used to control the flow of computation, differ from other program segments from a hierarchical point of view. So we introduce the new concepts called meta-flow in this paper.

Then the important question is how to formalize the interpretation of meta-flow. So we will try to formalize it by means of automata as defined below. Here we divide the elements of a meta-program segment into two kinds--i.e., when the flow of computation reaches meta-flow, one kind gives a certain input to automata and the other branches according to the state of automata. Next, we formally define program schemata with meta-flow.

§ 2 Program Schemata with Meta-flow

2. 1. Definition of the Model

Definition 2. 1: A program schema which is augmented with meta-flow is defined by the following seven-tuple.

$\alpha = \langle F, P, E, M, W, B, \Gamma \rangle$

Where

$F = \{f_1, f_2, \ldots, f_m\}$ is a finite set of operator symbols,

$P = \{p_1, p_2, \ldots, p_n\}$ is a finite set of predicate symbols,

$Q = \{p_1, \overline{p_1}, p_2, \overline{p_2}, \ldots, p_n, \overline{p_n}\}$ is a finite set of interpreted predicate symbols,

$E = \{IN, OUT\}$ consists of starting node IN and terminating node OUT,

$M = \langle X, S, \delta, s_0 \rangle$ is a machine model which interprets meta-program segment and the four-tuple is defined as follows;

X is a finite set of input symbols,

S is a set of states,

$\delta: S \times X \to S$ is a state transition function,

$s_0 \in S$ is an initial state,

$W = \{w(x_1), \ldots \mid x_i \in X\}$ shows inputs to the machine M, as defined above, where $w(x_i)$ means that input to M is x_i.

$B = \{B(s_1), B(s_2), \ldots \mid s_i \in S\}$ is a set of the expression of branching where $B(s_i)$ is defined as a mapping from S to (0, 1), and when the flow of program schemata reaches a node $B(s_i)$ and the state of M is not s_i, $B(s_i)$ is mapped to 0, however when the state of M is s_i, $B(s_i)$ is mapped to 1.

Γ shows the connective relation between nodes of flow

chart of program schema. The relation between a program schema and its fundamental one-dimensional expression is shown in Fig.2.1.

A Yanov's program schema β is generally defined to be $\beta = < F, P, E, \Gamma >$. So, in a program schema α with meta-flow, M, W and B are added to β. Next, we define two concepts called 'meta-program segment' and 'meta-flow'.

Definition 2.2 : Meta-program segments are the ones which are constructed by erasing all elements of $F \cup Q$ from a given program schema α. Namely, W and B are elements of meta-progam segments.

When the flow of a program reaches a node $B(s_i)$, $B(s_i)$ selects either of the two branches according to the state of the machine M at that time. Then, meta-flow is the flow of meta-program segments settled by a branch of $B(s_i)$.

By means of a program schema with meta-flow, we will provide several conveniences for some purposes. For example, while a program is being executed, according to the flow of computation in a certain part of a program we can control the flow of the other parts of it. In particular when several parts of program have some similarities, we can reduce them to the same one part and make the modified program play the same role as the original program by using meta-program segments properly. Namely, we can regard this model of a program as

one in which only particular nodes are assigned the fixed interpretation before execution.

2. 2. Equivalence Problem

Next we consider the extended definition of usual strong equivalence. As a preparation, we define two mapping functions f_Σ and g_Σ at first.

Definition 2.3 : Two mapping functions f_Σ and g_Σ are homomorphic and are defined in the following manner.

Let

$$\Sigma = \Sigma_1 \cup \Sigma_2 ,$$

$$\Sigma_1 = F \cup Q ,$$

$$\Sigma_2 = B \cup W .$$

Then f_Σ is defined as follows.

if $x \in \Sigma_1$, $f_\Sigma (x) = \lambda$,

if $x \in \Sigma_2$, $f_\Sigma (x) = x$,

and if $x_1, x_2 \in \Sigma$, $f_\Sigma (x_1 \cdot x_2) = f_\Sigma (x_1) \cdot f_\Sigma (x_2)$;

g_Σ is a mapping such that

if $x \in \Sigma_1$, $g_\Sigma (x) = x$,

if $x \in \Sigma_2$, $g_\Sigma (x) = \lambda$,

and if $x_1, x_2 \in \Sigma$, $g_\Sigma (x_1 \cdot x_2) = g_\Sigma (x_1) \cdot g_\Sigma (x_2)$;

where λ is a null symbol.

Next we define the set of strings $M(\alpha)$. Intuitively speaking, it is constructed by elements of $F \cup Q$ during

program execution considering only the behavior of the machine M interpreting meta-program segments.

The formal definition is as follows.

Definition 2.4; Let Θ be any set of strings.

Then g_X is defined as follows.

If $f_\Sigma(\omega) \in \Theta$, $g_X(\omega) = g_\Sigma(\omega)$;

If $f_\Sigma(\omega) \notin \Theta$, $g_X(\omega) = \lambda$;

where ω is any string.

Let $L(\alpha)$ be the set of one-dimensional strings of program schemata α . Then, Θ is any set of strings which is interpreted by the machine M. Informally, the assertion that a string $\omega \in f_\Sigma(L(\alpha))$ is interpreted by a machine M means that if $\omega = W(x_i)W(x_j)\ldots B(s_\ell)\ W(x_k)\ldots B(s_m)\ldots$, starting from an initial state s of the machine M, $x_i x_j \ldots$ is fed to M, then the state should be s_1, and later when $x_k \ldots$ is fed to M, the state should be $s_m, \ldots$etc. Then the set of strings $M(\alpha)$ which is constructed by elements of $F \cup Q$ is defined as follows. $M(\alpha) = g_X(L(\alpha))$

The extended definition of usual strong equivalence is followed.

Definition 2.5: Any program schema α_1 is strongly equivalent to other program schema α_2 if and only if the following relation holds.

$$M(\alpha_1) = M(\alpha_2).$$

§ 3 Program Schemata with Meta-Flow Controlled by a Finite Automaton

In the previous Chapter, we made a general definition of a program schema with meta-flow. There, the machine M, which was regarded as the model of meta-program segments, was defined by both finite and infinite automata.

In this Chapter, we discuss the case in which the automata are restricted to finite state automata. Then we show that several decision problems can be decidable.

3.1. Definition of the Model

Here, we make a definition in order to discuss decision problems in the section 3.2 of this chapter.

Definition 3.1: In a program schema $\alpha = \langle F, P, E, M, W, B, \Gamma \rangle$ with meta-flow, the machine $M = \langle X, S, \delta, s_0 \rangle$ which interprets meta-program segments is defined as follows. In this case, the machine M is called a finite automaton.

$S = \{s_0, s_1, \ldots, s_\ell\}$ is a finite set of states. Then, δ (a state transition function) is defined in the following manner.

When $s_i, s_j \in S$, $x \in X$, $\delta(s_i, x) = s_j$.

In such a program schema as defined above, a set Θ which is interpreted by the machine M (Refer to Definition 2.4.) becomes a regular set. The explanation is as follows.

We consider the following finite automaton M'.

$M' = \langle X', S', \delta', q_0, q_t \rangle$

Where $X' = \{W(x_1), \ldots, W(x_i), B(s_0), B(s_1), \ldots, B(s_\ell), \lambda\}$

$S' = \{B(s_0), \ldots, B(s_\ell), q_0, q_t\}$, i.e., S' is a set of nodes which remain when the nodes '$F \cup Q \cup W$' are erased from a given program schema α.

q_0 is an initial state, i.e., corresponds to a starting node 'IN'.

q_t is a final state, i.e., corresponds to a terminating node 'OUT'.

Here, we consider a diagram D constructed in the following manner. When a program schema α is given, we erase the nodes $F \cup Q \cup W$ and the left nodes are connected according to the connective sequence of the schema α. Then, δ' is defined as follows.

(i) When q_i leads to q_j directly and between q_i and q_j, $W(x_k)$ does not exist in the schema α,

$$\delta'(\lambda, q_i) = q_j \quad . \quad (q_i, q_j \in S')$$

(ii) When only one $W(x_k)$ exists between q_i and q_j ,

$$\delta'(W(x_k), q_i)\ q_j \quad .$$

(iii) When $W(x_{k1}), \ldots,$ and $W(x_{kh})$ are between q_i and q_j,

$$\delta'(W(x_{kh}), q_i) = q_i \quad (h=1, \ldots, n-1)$$

$$\delta'(W(x_{kh}), q_i) = q_j \quad .$$

(iv) When a predicate p exists in the schema α, a non-

deterministic branch occures in the diagram D. At that time,

$$\delta'(x', q_i) = q_r$$

$$\delta'(x', q_i) = q_s \quad . \ (r \neq s) \quad (x' \in X')$$

$$(\ddot{V}) \quad \delta'(B(s_i), q_j) = q_j \ .$$

Then we consider any string $\omega \in f_\Sigma(L(\alpha))$. ω is accepted by the machine M when the initial state is q_0 and the final state is q_t. Namely the flow starts in 'IN' and terminates in 'OUT'. The set of strings which are accepted by the finite automaton defined above is regular.

That is, θ is a regular set.

A simple example of this program schema with meta-flow is shown in Fig.3.1.. In this example, the machine $M=\langle X, S, \delta, s_0 \rangle$ is as follows.

$X=\{0, 1\}$, $S=\{s_0, s_1, \ldots, s_n\}$,

$\delta(s_i, 0) = s_{i+1}$ $(i=0, 1, \ldots, n-1)$

$\delta(s_j, 1) = s_{j-1}$ $(j=1, 2, \ldots, n)$

As shown in this example, it is very easy to express a program schema such that a certain segment in the flow of the program loops n-times, and when the flow reaches the other segment with a loop, make it loop n-times if we use this model.

3.2. Decision Problems

In this section, we will show that the following decision problems become decidable in such a program schema with meta-flow as defined above.

Theorem 3.1. : A program schema with meta-flow, which is controlled by finite state automata, is translatable into a regular program schema which is equivalent to it and has no meta-flow.

Proof : In a given program schema with meta-flow as defined above, we consider a program schema the nodes of which are correspond to a pair (a, b) -where a $\epsilon F \cup P$, b ϵS- according to the fixed interpretation of meta-program segments. In this case, the number of states are finite and so a given program can certainly be translated into such a direct-product type program schema. Any graph schema can be expressed as an equivalent regular program schema. Therefore if a given program schema with meta-flow can be transformed into a direct program schema with no meta-flow (which is practically a graph schema), it is also possible to express it as an equivalent regular program schema.

(Q. E. D.)

In this theorem, it is shown that a program schema with meta-flow controlled by finite automata can be transformed into an equivalent program schema with no meta-flow. But in a program schema with no meta-flow, the number of nodes will generally be much larger than a program schema with meta-flow. Also from this point of view, we know that it is useful to express any algorithm by means of a program schema with meta-flow.

From Theorem 3.1, and that an equivalence problem is decidable in a regular program schema, we can show the following theorem.

Theorem 3.2. : If two program schemata α_1 and α_2 with meta-flow controlled by finite state automata are given, it is derivable whether they are strongly equivalent or not.

Next, we consider the some properties of this model.

Definition 3.2. : A program schema with meta-flow, in which the state transition function δ of the machine M is not defined, is called a variable program schema with meta-flow. Then, we consider the case in which if δ in a variable program schema α_1 with meta-flow is determined appropriately, the variable program schema α_1 with meta-flow becomes equivalent to a certain program schema α_2 with meta-flow. In such a case, we would say that α_1 includes α_2 .

This concept is one example of the applications of meta-flow and it is important when we make one program play the same role as several programs.

Then we arrive at the following theorem.

Theorem 3.2. : It is decidable whether a variable program schema α_1 with meta-flow controlled by finite state automata includes a program schema α_2 with meta-flow controlled by

finite state automata or not. Moreover, when a finite number of program schemata α_1, α_2, ..., α_k with meta-flow controlled by finite state automata are given, we can make a variable program schema with meta-flow controlled by finite state automata which includes all the program schemata α_1,...., α_k.

Proof : In the case when a variable program schema α_1 with meta-flow controlled by finite automata is given, the number of states and also the number of inputs is finite. A state transition function δ is thought as the mapping function from $(s_i \times x_j)$ to s_k $(s \in S,\ x \in X)$. Then, the number of states and inputs is finite, so the number of δ's, which are defined according to each permutation (s_i, x_j, s_k), is finite. In consequence, the number of possible program schemata, which are determined by a variable program schema α_1, is also finite. From Theorem 3.1., we can know that these program schemata are translatable into regular program schemata.

Here, a given program schema α_2 is also translatable into a regular program schema. An equivalence problem between regular program schemata is decidable, so it is decidable to determine whether or not a variable program schema α_1 includes a program schema α_2.

The proof of the latter part of Theorem 3.3 is elementary. Namely, we can combinate several given program schemata $\alpha_1, \alpha_2, \ldots, \alpha_k$ with meta-flow controlled by finite automata in such a way that we make k-branches, using k new predicates in additional meta-program segments. (Q.E.D.)

§ 4 Program Schemata with Meta-flow Controlled by a Push-down Stack

In this chapter, we consider the case in which the meta-program segments are interpreted by a push-down stack. A decision problem in this model is generally undecidable. But certain restricted classes of this model are decidable. Moreover, in this model of program schemata, it is decidable to determine whether or not there exists an equivalent regular program schema.

From now on, we abbreviate program schemata with meta-flow controlled by a push-down stack as program schemata with a push-down stack.

We will have several practical applications of this model of program schemata with a push-down stack. For example, we can use this model when we want to make a flow chart of the algorithm which computes a recursion equation, not directly expressed in an iterative form. Furthermore, we can also use it as the model of subroutine-call.

4.1. Definition of the Model

First, we define a model of program schemata with a push-down stack in this section. Then, we show one example

of this model.

Definition 4.1: In a program schema $\alpha=<F, P, E, M, W, B, \Gamma>$ with a push-down stack, the machine $M=<X, S, \delta, S_0^{\#}>$, B and W are defined as follows.

$X = \{x_1, x_2, \ldots, x_i\}$ is a finite set of input symbols.

S shows the contents of the push-down stack.

δ is defined in the following manner.

$\delta_W(S, x_i)=Sx_i$ when the flow reaches $W(x_i)$. Namely, δ_W is the function which shows that we push down an entry x_i into the most upper part of a push-down stack.

$\delta_B(Sx_i, x_i)=S$ .... (i)

$\delta_B(Sx_j, x_i)=Sx_j (i \neq j)$...(ii) when the flow reaches $B(X_i)$. Namely, δ_B is the function which shows that we pop up an entry X_i from the upper part of the push-down stack if the entry of the upper part of the push-down stack is X_i, or we do not change the state of the push-down stack if the entry of the upper part of the push-down stack is not X_i.

$B(x_i)$ is mapped to '1' when the action of (i) occurs, or is mapped to '0' when the action of (ii) occurs.

In this case, it is difficult to give a formal definition of a set θ . Therefore we adopt an alternative approach to seek directly $M(\alpha)$.

When a program schema α with a push-down stack are given, we define a grammar $G_\alpha = \langle V, F \cup Q \cup \#, D, S_0^{\#} \rangle$ in the following manner. Where, D is a set of rewriting rules, $S_0^{\#}$ is an initial symbol of a push-down stack, $V = \{x_1, \ldots, x_i, S_1, \ldots, S_m, S_0^{\#}\}$ is a set of non-terminal symbols, $F \cup Q \cup \#$ is a set of terminal symbols, and # is an end mark.

D is constructed as follows;

(1) An edge, which leads away from a node 'IN' in this program schemata, is attached a label '$S_0^{\#}$'.

(2) We consider an edge, labelled '0', which leads away from a node corresponding to $B(x_i)$ and which does not lead to a node corresponding to $B(x_j)$ or a node 'OUT'. We assume that $\{i_1, \ldots, i_h\}$ is a set of such edges as defined above. Then a certain edge i_j in a set $\{i_1, \ldots, i_h\}$ is attached a label '$S_j^{\#}$' $(1 \leq j \leq h)$.

(3) From all edges in a given program schema, we exclude an edge, which leads to a node corresponding to $B(x_i)$, $W(x_j)$, or a node OUT; or leads away from a node corresponding to $B(x_i)$ and is labelled '1'. We assume that $\{j_1, j_2, \ldots, j_h\}$ is such a set of edges. Then a certain edge j_i in the set is attached a label 'S_i'.

(4) To each segments in a given program schema, shown in Fig. 4.1, we apply each rewriting rules shown directly below Fig. 4.1.

(5) When we erase an edge, labelled 'S_i#' according to the rule of (2), from a given program schema, we have several separate section-graphs. We define each section-graph

as a 'block'. We assume that an edge, which leads to a certain block, is being attached a label '$S_i^{\#}$' and an edge, which leads away from the block, is being attached labels 'S_k' and '$S_j^{\#}$'. Then, when the flow of computation reaches the exit edge of the block, we apply the following rewriting rule:

$$S_i^{\#} \rightarrow S_k S_j^{\#}.$$

Refering to Fig.4.2. We note that in Fig.4.2, Fig.5.3 and Fig.4.5, the section enclosed in the dotted line shows a certain block.

(6) We assume that an edge, which leads to a certain block, is being attached a label '$S_i^{\#}$' and that an edge, which leads away from the block, is being attached a label '$S_j^{\#}$' and the exit edge of the block leading to a node $W(x_h)$ is labelled 'S_k'. When the flow of computation reaches the exit edge of a node $W(x_h)$, we apply the following rewriting rule.

$$S_i^{\#} \rightarrow S_k x_h S_j^{\#}$$ (Refer to Fig.4.3.)

(7) We consider an edge which leads away from a node corresponding to $B(x_i)$ mapped to '1'. (Refer to Fig.4.4.)

(i) When the edge leads directly to a node corresponding to $B(x_j)$ or $W(x_j)$, we apply two null operators λ between a node corresponding to $B(x_i)$ and a node corresponding to $B(x_j)$ or $W(x_j)$. (In this case i = j is included.)

(ii) When the edge leads to a node corresponding to $B(x_j)$ or $W(x_j)$, only one operator f or predicate p intervening between them, we assume that there is one null operator λ between $B(x_i)$ and $B(x_j)$, or $B(x_i)$ and $W(x_j)$.

(iii) When a node corresponding to $W(x_i)$ leads directly to a

node corresponding to $W(x_j)$, we assume that a null operator exists between them.

(8) When an operator f leads to a node 'OUT' and a label of the edge which leads to the operator f is 'S_i', we use the following rewriting rule.

$S_i \rightarrow f$.

When a predicate p leads to a node 'OUT', we use the following rewriting rule according to the semantics of p.

$S_i \rightarrow p$, or $S_i \rightarrow \overline{p}$.

(9) When a block leads to a node 'OUT' and a label of the edge which leads to the block is $S_i^{\#}$, we use the following rewriting rule at the time when the flow of computation reaches a node 'OUT'. (Refer to Fig. 4.5.)

$S_i^{\#} \rightarrow \#$.

According to the flow of computation, we construct a grammar G_α in such a way as defined above. Then we apply the grammar G_α according to the flow of computation. Consequently, a set of one-dimensional strings of $F \cup Q$ is produced. Here, we define the set to be $L(G_\alpha)$. From the definition of $M(\alpha)$ in section 2.2. and the definition of $L(G_\alpha)$, we obtain the following theorem.

Theorem 4.1 : The following relation exists between a program schema α with a push-down stack and its grammar G_α.

$L(G_\alpha) = M(\alpha)\#$.

From the above theorem a strong equivalence problem between a program schema α_1 and α_2 with meta-flow controlled by a push-down stack can be reduced to an equivalence problem between $L(G_{\alpha 1})$ and $L(G_{\alpha 2})$.

In Fig. 4.6, we show an example of this model of a program schema with a push-down stack. In this model, a grammar G_α is as follows.

$G_\alpha=(V, F\vee Q\vee\#, D, S_0{}^\#)$

where $V=\{S_1,S_2,S_3,S_4,S_5,S_6,S_7,S_0{}^\#,S_1{}^\#,x\}$

$F=\{f_1,f_2,f_3,f_4,f_5\}$

$Q=\{p\}$

The set of rewriting rule D is

$S_1\to pS_2$, $S_1\to\bar{p}S_4$, $S_2\to f_1S_3x$, $S_3\to pS_2$, $S_3\to\bar{p}S_4$, $S_4\to f_2$, $x\to f_3$, $S_0{}^\#\to S_6S_1{}^\#$, $S6\to f_4S_7$, $S_7\to f_5$, and $S_1{}^\#\to\#$.

We show one example of a string $L(G_\alpha)$, produced by means of G_α which is above defined.

$S_1S_0{}^\# \to pS_2S_0{}^\# \to pf_1S_3xS_0{}^\# \to pf_1pS_2xS_0{}^\# \to pf_1pf_1S_3xxS_0{}^\#$

$\to pf_1pf_1\bar{p}S_4xxS_0{}^\# \to pf_1pf_1\bar{p}f_2xxS_0{}^\# \to pf_1pf_1\bar{p}f_2f_3xS_0{}^\#$

$\to pf_1pf_1\bar{p}f_2f_3f_3S_0{}^\# \to pf_1pf_1\bar{p}f_2f_3f_3S_6S_1{}^\#$

$\to pf_1pf_1\bar{p}f_2f_3f_3f_4S_7S_1{}^\# \to pf_1pf_1\bar{p}f_2f_3f_3f_4f_5S_1^\#$

$\to pf_1pf_1\bar{p}f_2f_3f_3f_4f_5\#$.

Next, we show that in a restricted class of this model of program schemata with a push-down stack, it becomes decidable whether or not two program schemata are equivalent and whether or not there exists a regular program schema equivalent, to a given program schema with a push-down stack.

4.2 S-program Schemata and Decision Problems

A set of strings $L(G_\alpha)$, as explained in the previous sections, is generally a set of context-free languages. In this section, we define S-program schemata which are general program schemata with a push-down stack defined in the section 4.1 but with some limitations.

Definition 4.2 : An S-program schema is defined to be a program schema with a push-down stack which also satisfies the following restrictions.

(1) If there exist one more nodes corresponding to $B(x_i)$ in a given program schema, each element which is led to by each edge leading away from each node corresponding to $B(x_i)$ and being in the label of '1', should be different.

(2) The element should be an element of $F \cup Q$.

Definition 4.3. : A language is called an S-language if the following condition is satisfied. Here, the S-languages are generated by an S-grammar.

"Each rewriting rule in an S-grammar has the form $V \rightarrow xV_1 \ldots Vn$, $n \geq 0$, and the pairs(V,x) are distinct among the rules."($V_1, V_2, \ldots, Vn$, and V_i are all terminal symbols. x is a non-terminal symbol and $x \neq \lambda$.)

From these two definitions and the rewriting rules explained before, we can derive the following theorem.

Theorem 4.2. : If a given program schema α is an S-program schema, a set of strings $L(G_\alpha)$ is an S-language.

Proof : In Definition 4.2, we place limitations on a general program schema with a push-down stack so that $L(G_\alpha)$ generated by means of the rewriting rules becomes an S-language in Definition 4.3. So, it is clear that Theorem 4.2. holds. (Q.E.D.)

From Reference(11), we know that it is decidable to determine whether or not a given S-language is regular and that an algorithm exists to determine whether two given S-grammars generate the same language. So, taking the above theorem into consideration, we can derive the following two theorems.

Theorem 4.3. : When an S-program schema α is given, it is decidable to determine whether or not there exists a regular program schema equivalent to it.

Proof : A context-free grammar is said to be self-embedding if there is some $Z \in Vn$, and some $\alpha, \beta \neq \varepsilon$ such that $Z \rightarrow \alpha Z \beta$. If any grammar for an S-language is not self-embedding(which is clearly decidable), then an S-language is regular. The converse is also true for S-grammars. In consequence, we can determine the above problem if we know whether the S-grammar is self-embedding. (Q.E.D.)

Theorem 4.4. : When two program schemata α_1 and α_2 are given, it is decidable to determine whether they are strongly equivalent or not.

Outline of proof : The detailed proof is much similar to the proof in Ref.(11). Here, we show the outline of proof.

If, in some context-free grammar G, and for some Z,Y $\in$Vn, Z $\Rightarrow$x if and only if Y$\Rightarrow$ x, x $\in$Vt*, then write Z$\equiv$Y. That is, Y is equivalent to Z if and only if they generate the same terminal strings according to a given grammar. (if $\alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha n$, $n \geq 1$, we write $\alpha_1 \Rightarrow {}^{\alpha}n$)

Let the grammars G_1 and G_2, corresponding to α_1 and α_2, with starting symbols S_1 and S_2, respectively. By proceeding as described in Ref.(11), this equivalence pair(i.e. S_1 and S_2) can be replaced by new pairs, upon which the equivalence of G_1 and G_2 depends. We will show that the set of equivalences generated by iterating this procedure is finite, and that if G_1 is not equivalent to G_2, the shortest terminal string in which they differ will be indicated during this process. If no conflicts are found, then we may conclude that $L(G_1)=L(G_2)$. (Q.E.D.)

References

(1) J.W.de Bakker, Semantics of Programming Languages, in "Advance in Computers" (F.L.Alt and M.Rubinoff, eds.), Vol.8, pp.173-227,Academic Press, New York and London (1967)

(2) C.Böhm and G.Jacopini, Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules, Comm. Assoc. Computing Machinery 9, pp.366-372 (1966)

(3) D.C.Cooper, On the Equivalence of Certain Computations, Comp. J. 9, pp.45-52 (1966)

(4) D.C.Cooper, Some Transformations and Standard Forms of Graphs, with Applications to Computer Programs, in"Machine Intelligence"(E.Dale and D.Michie, eds.)Vol.2, pp.21-32, Oliver and Boyd,Edinburg (1967)

(5) R.W.Floyd, Assigning Meaning to Programs, in"Mathematical Aspects of Computer Science," Proc. of Symposia in Applied Mathematics, Vol.19 (J.T.Schwartz, ed.) pp.19-32, American Mathematical Society, Providence, Rhode Island (1967)

(6) S. Igarashi, On the Logical Schemes of Algorithms (in Japanese),Information Processing in Japan 3,pp.12-18(1963)

(7) T. Ito, Some Formal Properties of a Class of Program Schemata, Proc. IEEE Symposium on Switching and Automata Theory (1968)

(8) L.A.Kaluzhnin, Algorithmization of Mathematical Problems, in"Problems of Cybernetics," Vol.2, pp.371-391, Pergamon Press, New York (1961)

(9) D.M.Kaplan, Regular Expression and the Equivalence of Programs, J. Computer and System Science, Vol.3 (1969)

(10) R.M.Karp, A Note on the Application of Graph Theory to Digital Computer Programming, Information and Control 3, pp.179-189 (1960)

(11) A.J.Korenjak and J.E.Hopcroft, Simple Deterministic Languages, IEEE Conf. on Switching and Automata Theory, (1966)

(12) D.C.Luckham, D.M.R.Park and M.S.Paterson, On Formalized Computer Programs, J. Computer and System Science, Vol.4,(1970)

(13) Z.Manna and A.Pnueli, Formalization of Properties of Functional Programs, J. ACM 17, 555-569 (1970)

(14) J.McCarthy, A Basis for a Mathematical Theory of Computation, in " Computer Programming and Formal Systems" (P.Braffort and D.Hirschberg, eds.), pp.33-69, North Holland Publishing Co., Amsterdam (1963)

(15) J.McCarthy, Towards a Mathematical Science of Computation, in "Information Processing 1962," Proc. IFIP Congress 1962(C.M. Popplewell, ed.) , pp.21-28, North Holland Publishing Co., Amsterdam(1963)

(16) J.McCarthy and J.Painter, Correctness of a Compiler for Computer Science," Proc. of Symposia in Applied Mathematics, Vol.19 (J.T.Schwartz, ed.), pp.33-41, American Mathematical Society, Providence, Rhode Island (1967)

(17) I.Okinaka, Y.Kambayashi, A.Sha and T.Kiyono, On the Program Schemata with Push-down Stack (in Japanese), Rec. of the 11th Convention of Information Processing Society of Japan, No.88, (1970)

(18) M.S.Paterson, Equivalence Problems in a Model of Computation, Doctoral Dissertation, Cambridge University (1967)

(19) M.S.Paterson, Program Schemata, Artificial Intelligence, Vol.3, (1968)

(20) J.D.Rutledge, On Yanov's Program Schemata, J. Assoc. Computing Machinery 11, pp.1-9 (1964)

(21) A.Salomaa, Two Complete Axiom Systems for the Algebla of Regular Events, J. Assoc. Computing Machinery 13, pp.158-169 (1966)

(22) A.Shurmann, The Application of Graphs to the Analysis of Distribution of Loops in a Program, Information and Control 7, pp.275-282 (1964)

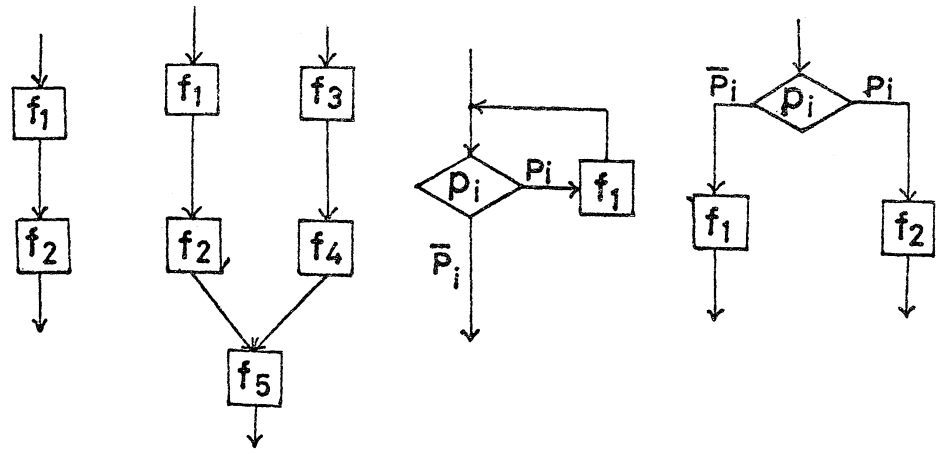
(23) A.Sha, Y.Kambayashi, I.Okinaka and T.Kiyono, On the Program Schemata with Meta-flow (in Japanese), Rec. of the 11th Convention of Information Processing Society of Japan,No.87(1970)

(24) H.R.Strong,Jr. Translating Recursion Equations into Flow Charts, ACM Symp. on Theory of Computing (1970)

(25) Y.I.Yanov, On the Equivalence and Transformations of Program Schemes, Comm. Assoc. Computing Machinery 1 (10), pp.8-12(1958)

(26) Y.I.Yanov, On Matrix Program Schemes, Comm. Assoc. computing Machinery 1 (12), pp.3-6 (1958)

(27) Y.I.Yanov, The Logical Schemes of Algorithms, in"Problems of Cybernetics," Vol.1, pp.82-140, Pergammon Press, New York (1960)



$f_1 \cdot f_2$, $(f_1 \cdot f_2 \cup f_3 \cdot f_4) \cdot f_5$, $(p_i f_1)^* \cdot \bar{p}_i$ $\bar{p}_i \cdot f_1 \cup p_i \cdot f_2$

Fig. 2.1 One-dimensional expression of a program schema

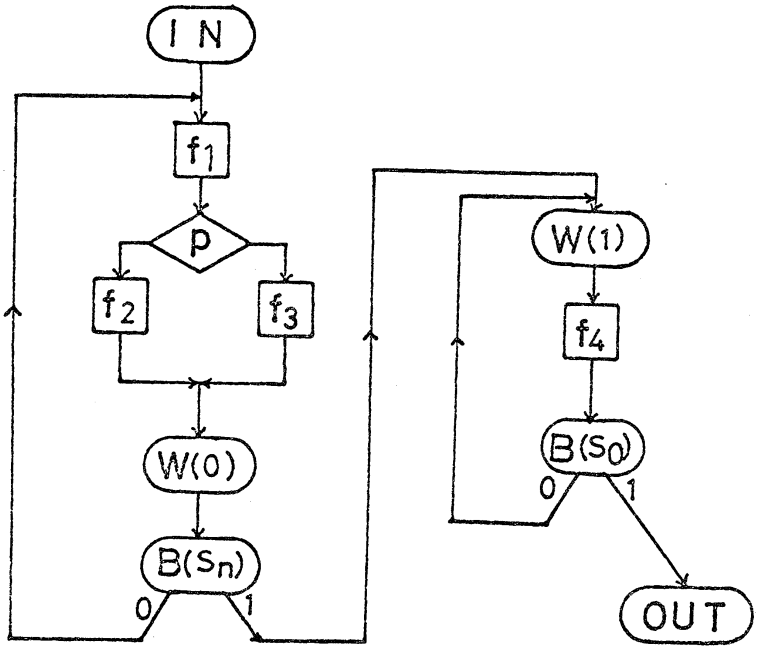


Fig. 3.1 An example of a program schema with meta-flow (finite automata)

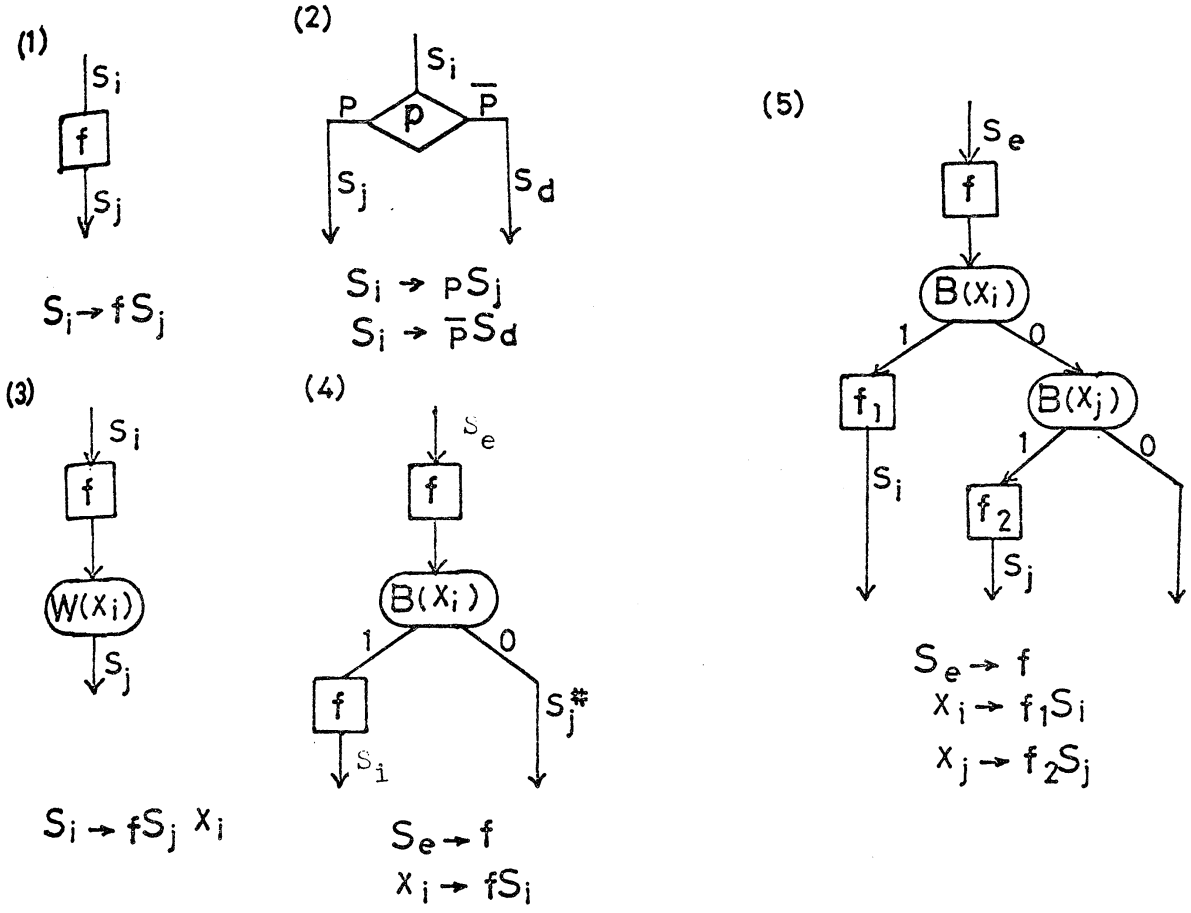
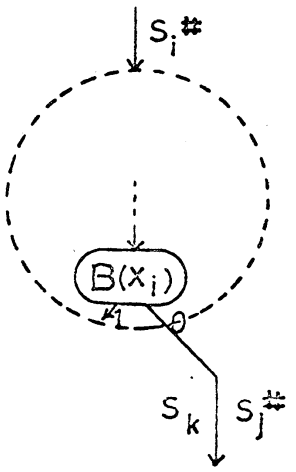
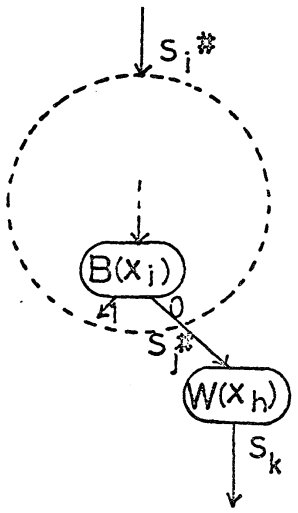


Fig. 4.1 Rewriting rules of a program schema with a push-down stack (No. 1)



$S_i^{\#} \rightarrow S_k S_j^{\#}$

Fig. 4.2
Rewriting rule (No.2)



$S_i^{\#} \rightarrow S_k x_h S_j^{\#}$

Fig. 4.3
Rewriting rule (No. 3)

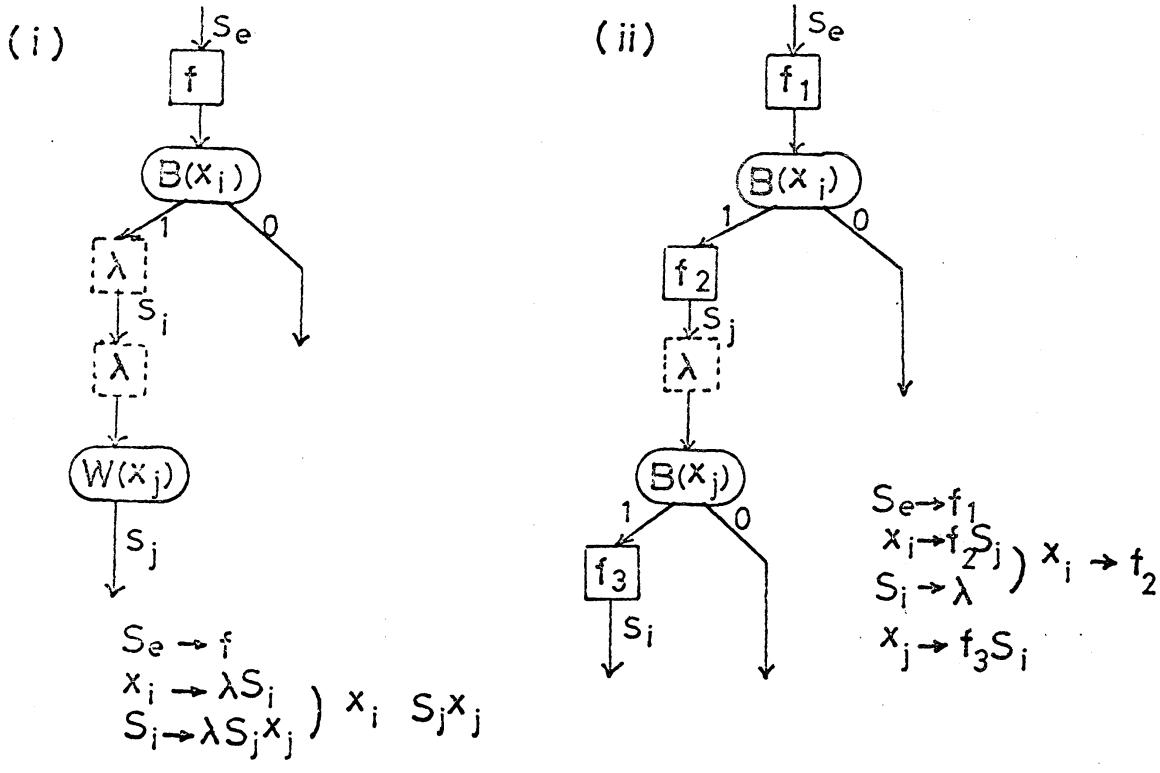


Fig. 4.4 Rewriting rules (No. 4)

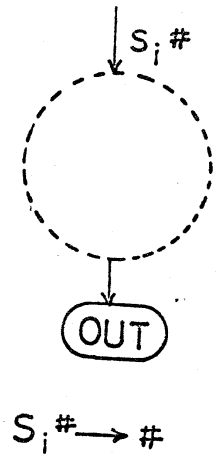


Fig. 4.5

Rewriting rule (No. 5)

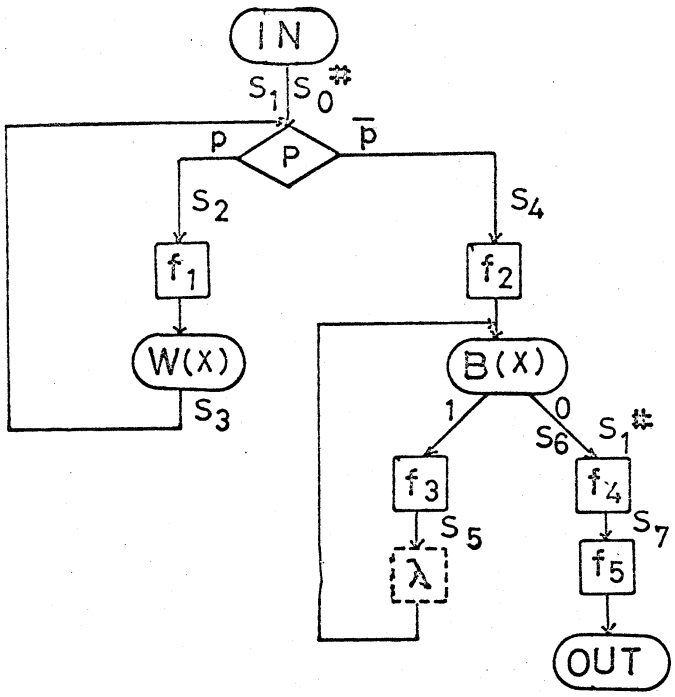


Fig. 4.6 An example of a program schema with a push-down stack